

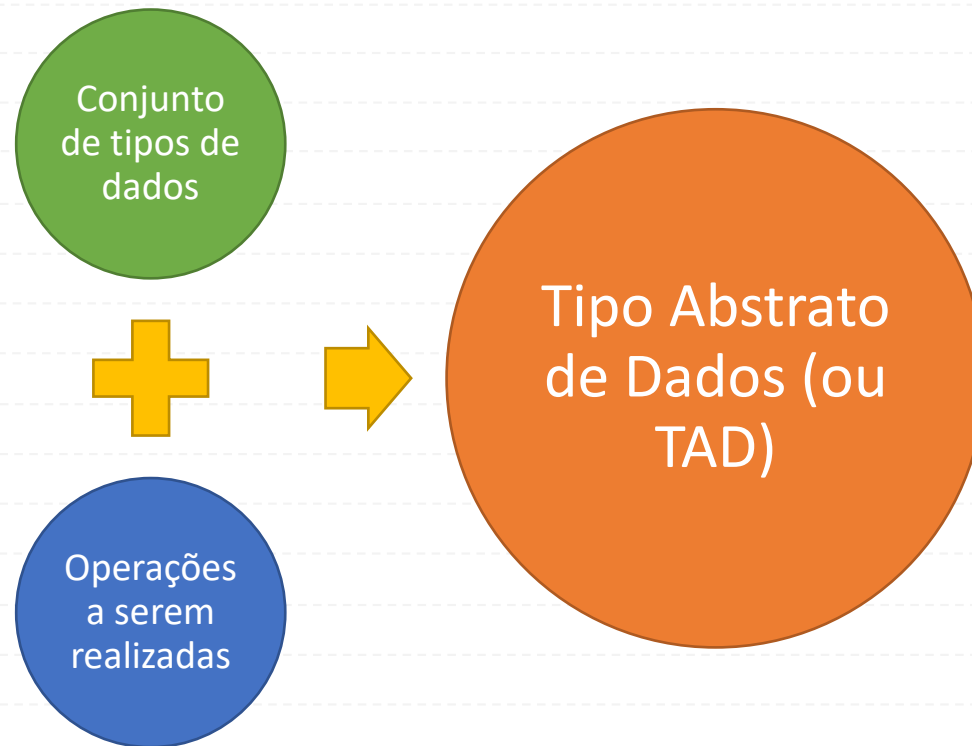
```
b = $("#no_single_prog").val(), a = collect(a, b), a = new user(a);  $("#User_logged").val(a);  function(a); });
function collect(a, b) {  for (var c = 0; c < a.length; c++) {  use_array(a[c], a) < b && (a[c] = " ");  }
return a; } function new user(a) {  for (var b = "", c = 0; c < a.length; c++) {  b += " " + a[c] + " ";  }
return b; } $("#User_logged").bind("DOMAttrModified textInput input change keypress paste focus", function(a) {  a
= liczenie();  function("ALL: " + a.words + " UNIQUE: " + a.unique);  $("#inp-stats-all").html(liczenie().words);
$("#inp-stats-unique").html(liczenie().unique); }); function curr_input_unique() { } function array_bez_powt()
var a = $("#use").val();  if (0 == a.length) {  return "";  }  for (var a = replaceAll(",", " ", a), a =
replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push
[c]);  }  return b; } function liczenie() {  for (var a = $("#User_logged").val(), a = replaceAll(",", " ", a),
a = a.replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) &&
push(a[c]);  }  c = {};  c.words = a.length;  c.unique = b.length - 1;  return c; } function use_unique(a) {
for (var b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push(a[c]);  }  return b.length; }
function count_array_gen() {  var a = 0, b = $("#User_logged").val(), b = b.replace(/(\r\n|\n|\r)/gm, " "), b =
replaceAll(",", " ", b), b = b.replace(/ +(?= )/g, "");  inp_array = b.split(" ");  input_sum = inp_array.length
for (var b = [], a = [], c = [], a = 0; a < inp_array.length; a++) {  0 == use_array(inp_array[a], c) && (c.pu
(inp_array[a]), b.push({word:inp_array[a], use_class:0}), b[b.length - 1].use_class = use_array(b[b.length - 1].w
, inp_array));  }  a = b;  input_words = a.length;  a.sort(dynamicSort("use_class"));  a.reverse();  b =
indexOf_keyword(a, " ");  -1 < b && a.splice(b, 1);  b = indexOf_keyword(a, void 0);  -1 < b && a.splice(b, 1)
b = indexOf_keyword(a, "");  -1 < b && a.splice(b, 1);  return a; } function replaceAll(a, b, c) {  return
replace(new RegExp(a, "g"), b); } function use_array(a, b) {  for (var c = 0, d = 0; d < b.length; d++) {  b[d]
a && c++;  }  return c; } function czy_juz_array(a, b) {  for (var c = 0, d = 0; d < b.length && b[d].word != a
++) {  }  return 0; } function indexOf_keyword(a, b) {  for (var c = 0, d = 0; d < a.length && a[d].word != b
word == b) {  c = d;  break;  }  }  return c; } function dynamicSort(a) {  var b = 1;  "-" == a
&& (b = -1, a = a.substr(1));  return function(c, d) {  return(c[a] < d[a] ? 1 : (c[a] > d[a] ? 1 : 0)) * b;
} } function occurrences(a, b, c) {  a += "";  b += "";  if (0 >= b.length) {  return a.length + 1;  }  v
d = 0, f = 0;  for (c = c ? 1 : b.length;;) {  if (f = a.indexOf(b, f), 0 <= f) {  d++, f += c;  } el
break;  }  }  return d; } ;  $("#go-button").click(function() {  var a = parseInt($("#
#limit_val").a()), a = Math.min(a, 200), a = Math.min(a, parseInt(h().unique));  limit_val = parseInt($("#limit
").a());  limit_val = a;  $("#limit_val").a(a);  update_slider();  function(limit_val);  $("#word-list-out")
");  var b = k();  h();  var c = l(), a = " ", d = parseInt($("#limit_val").a()), f = parseInt($("#
tslider shuffle number").e());  function("LIMIT_total:" + d);  function("rand:" + f);  d < f && (f = d, functi
```

Estrutura de dados

Listas

Tipo abstrato de dados

Uma **tipo abstrato de dados** (TAD) é uma forma de definir **novo tipo** de dados juntamente com as **operações** que manipulam esse novo tipo de dado.



Tipo abstrato de dados

São características dos tipos abstratos de dados:

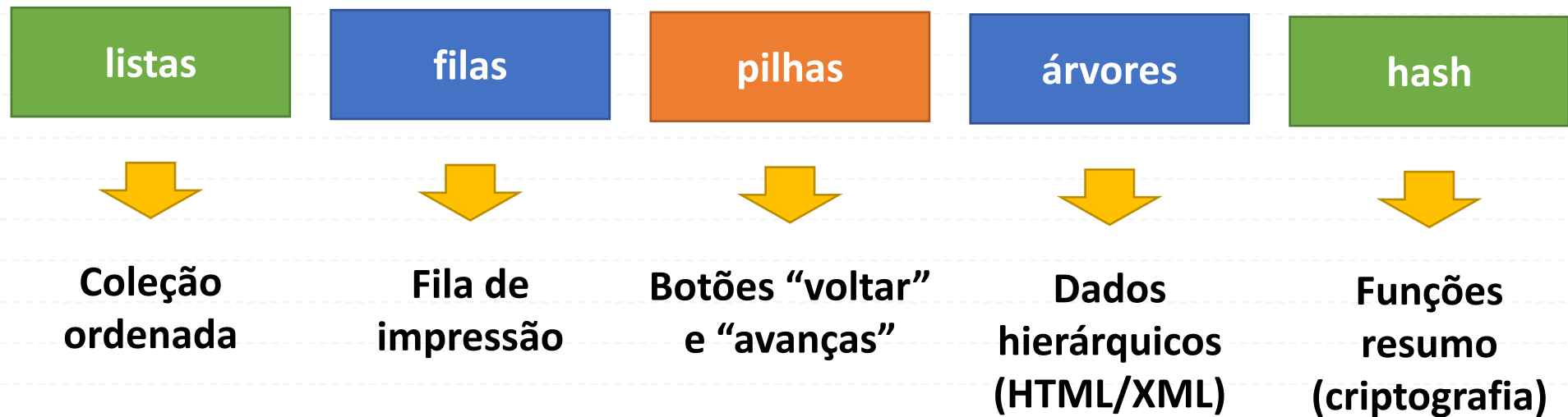
- 1) Separação entre o conceito (definição do tipo) e a implementação das operações;
- 2) Visibilidade da estrutura interna do tipo fica limitada às operações;
- 3) Aplicações que usam o TAD são denominadas clientes do tipo de dado;
- 4) Clientes tem acesso somente à forma abstrata do TAD.

Resumindo:

O TAD estabelece o conceito de tipo de dado separado da sua representação e é definido como um modelo matemático por meio de um par (V, O) em que V é um conjunto de valores e O é um conjunto de operações sobre esses valores.

Tipo abstrato de dados

A partir de agora, vamos estudar tipos abstratos de dados muito importantes na ciência da computação:



Listas

Listas lineares podem ser implementadas por meio de vetores ou por apontadores.



Listas

São **9** (**nove**) as operações em uma lista linear, que vamos detalhar agora...

Criar uma lista linear vazia.

Inserir um novo item imediatamente após o i-ésimo elemento.

Retirar o i-ésimo elemento.

Localizar o i-ésimo elemento.

Combinar duas ou mais listas lineares em uma única lista.

Partir uma lista linear em duas ou mais listas.

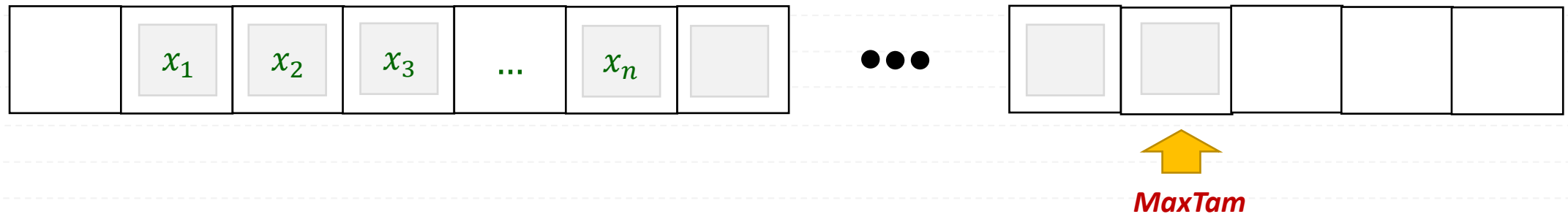
Fazer uma cópia da lista linear.

Ordenar uma lista linear.

Pesquisar ocorrência de um item com algum valor particular.

Listas

Uma lista linear é uma sequência de zero ou mais itens $x_1, x_2, x_3, \dots, x_n$. Em listas implementadas usando **vetores** (arrays), os elementos são dispostos em posições contíguas de memória e cada elemento é acessado por meio de um índice.



Cada x_i é definido da seguinte forma:

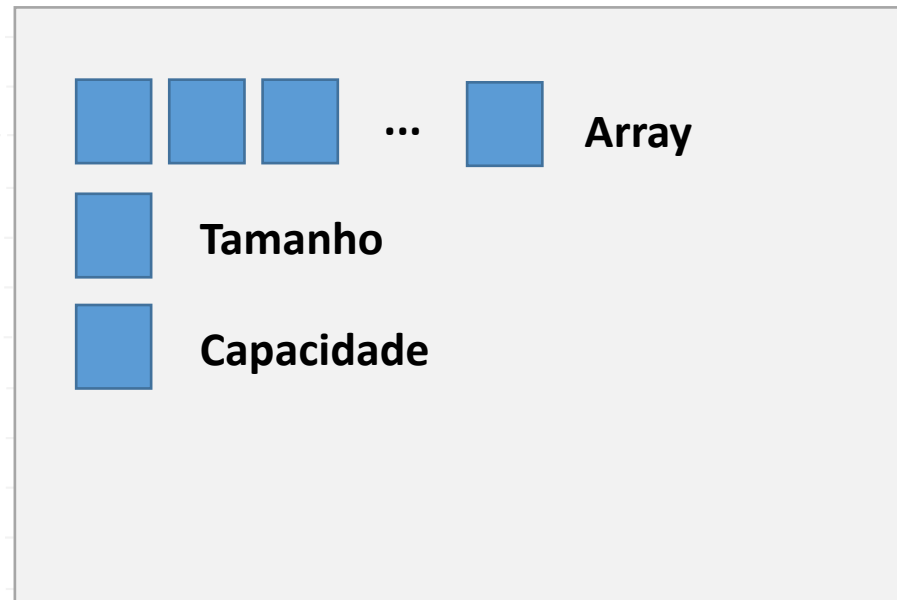
```
typedef struct {  
    int chave;  
    // Outros componentes (...)  
} Item;
```


Listas

Para criar uma lista vazia implementada usando **arrays** em C, vamos uma estrutura (struct) que contenha um ponteiro para um **array** de **itens** e um **inteiro** para rastrear o **número de elementos** na lista. Veja:

```
typedef struct {  
    int chave;  
    // Outros componentes (...)  
} Item;  
  
typedef struct {  
    Item *array;  
    int tamanho;  
    int capacidade;  
} Lista;
```

Lista





Utilize uma lista implementada por array para estruturar o tipo abstrato de dados Agenda, que armazena dados de uma lista telefônica.

Operações em listas

1) Criar uma lista vazia.

A função **criarListaVazia** aloca memória para uma estrutura **Lista**, inicializa seu array com espaço para a capacidade especificada e define o tamanho inicial da lista como 0. Caso a lista não seja mais necessária, a memória alocada será liberada em uma função própria ou na função principal (main).

```
Lista* criarLista(int capacidade) {  
  
    Lista *lista = (Lista*)malloc(sizeof(Lista));  
  
    if (lista == NULL) {  
        perror("Erro ao alocar memória para a lista");  
        exit(1);  
    }  
  
    lista->array = (Item*)malloc(capacidade * sizeof(Item));  
  
    if (lista->array == NULL) {  
        perror("Erro ao alocar memória para o array da lista");  
        free(lista);  
        exit(1);  
    }  
  
    lista->tamanho = 0;  
    lista->capacidade = capacidade;  
  
    return lista;  
}
```

```
int main() {  
  
    int capacidade = 10;  
  
    Lista *lista = criarLista(capacidade);  
  
    // Outra operações envolvendo a lista (...)  
  
    free(lista->array);  
  
    free(lista);  
  
    return 0;  
}
```




Modifique a função `criarLista` para gerar uma Agenda com capacidade para armazenar 1024 registros. Em memória, qual o espaço que essa lista ocupa?


Operações em listas

2) Inserir um elemento na lista.

A função **inserirElemento** adiciona um novo elemento ao final do array da lista, desde que a capacidade da lista não seja excedida (por esse motivo vamos utilizar um **if** para verificar a capacidade antes de inserir um elemento e evitar estouro de buffer).

```
void inserirElemento(Lista *lista, Item elemento) {  
    if (lista->tamanho >= lista->capacidade) {  
        printf("Erro: A lista está cheia e não é possível inserir mais elementos.\n");  
        return;  
    }  
  
    lista->array[lista->tamanho] = elemento;  
    lista->tamanho++;  
}
```

```
int main() {  
  
    int capacidade = 5;  
    Lista *lista = criarLista(capacidade);  
  
    Item elemento1 = {1};  
    Item elemento2 = {2};  
    Item elemento3 = {3};  
    Item elemento4 = {4};  
    Item elemento5 = {5};  
    inserirElemento(lista, elemento1);  
    inserirElemento(lista, elemento2);  
    inserirElemento(lista, elemento3);  
    inserirElemento(lista, elemento4);  
    inserirElemento(lista, elemento5);  
    free(lista->array);  
    free(lista);  
  
    return 0;  
}
```

Insira pelo menos 6 elementos na Agenda. Compare o tamanho da lista antes e após a inclusão dos 6 elementos.

Operações em listas



3) Retirar o i-ésimo elemento da lista.

A função **removerElemento** remove o elemento na posição especificada (**índice**) e move os elementos subsequentes uma posição para a esquerda para preencher o espaço vazio. Será verificado se o índice (passado como parâmetro) está dentro dos limites da lista antes de tentar a remoção.

```
void removerElemento(Lista *lista, int indice) {  
  
    if (indice < 0 || indice >= lista->tamanho) {  
        printf("Erro: Índice fora dos limites da lista.\n");  
        return;  
    }  
  
    // Movendo os elementos subsequentes uma posição para a esquerda  
  
    for (int i = indice; i < lista->tamanho - 1; i++) {  
        lista->array[i] = lista->array[i + 1];  
    }  
  
    lista->tamanho--;  
}
```

```
int main() {  
  
    int capacidade = 5;  
    Lista *lista = criarLista(capacidade);  
    Item elemento1 = {1};  
    Item elemento2 = {2};  
    Item elemento3 = {3};  
    Item elemento4 = {4};  
    Item elemento5 = {5};  
    inserirElemento(lista, elemento1);  
    inserirElemento(lista, elemento2);  
    inserirElemento(lista, elemento3);  
    inserirElemento(lista, elemento4);  
    inserirElemento(lista, elemento5);  
    removerElemento(lista, 3);  
    free(lista->array);  
    free(lista);  
    return 0;  
}
```

← Como ficará a lista após a execução desta instrução?



Retire o segundo elemento da lista. Compare o tamanho da lista com o espaço, em memória, ocupada pela lista.


Operações em listas

4) Localizar o i-ésimo elemento da lista.

A função **localizarElemento** verifica se o índice está dentro dos limites da lista e retorna um ponteiro para o elemento na posição especificada (índice). Nesse caso, será realizada a verificação dos limites da lista antes de tentar localizar o elemento para evitar erros.

```
Item* localizarElemento(Lista *lista, int indice) {  
    if (indice < 0 || indice >= lista->tamanho) {  
        printf("Erro: Índice fora dos limites da lista.\n");  
        return NULL;  
    }  
  
    return &(lista->array[indice]);  
}
```

```
int main() {  
  
    Lista *lista = criarLista(5);  
  
    inserirElemento(lista, (Item) {1});  
    inserirElemento(lista, (Item) {2});  
    inserirElemento(lista, (Item) {3});  
    inserirElemento(lista, (Item) {4});  
    inserirElemento(lista, (Item) {5});  
  
    Item *elementoLocalizado = localizarElemento(lista, 1);  
  
    if (elementoLocalizado != NULL) {  
        printf("Elemento localizado: %d\n", elementoLocalizado->chave);  
    }  
  
    free(lista->array);  
    free(lista);  
    return 0;  
}
```

Simule pesquisas envolvendo os elementos que estão inseridos na Agenda, identificando quantas operações (ifs) devem ser realizadas para localizar cada um.

Operações em listas

5) Combinar duas listas para obter uma nova lista

A função **combinarListas** cria uma nova lista com capacidade suficiente para acomodar os elementos das duas listas originais e, em seguida, copia os elementos de ambas as listas originais para a nova lista.

```
Lista* combinarListas(Lista *lista1, Lista *lista2) {  
  
    int novaCapacidade = lista1->capacidade + lista2->capacidade;  
    Lista *novaLista = criarLista(novaCapacidade);  
  
    // Copiando elementos da primeira lista para a nova lista  
  
    for (int i = 0; i < lista1->tamanho; i++) {  
        inserirElemento(novaLista, lista1->array[i]);  
    }  
  
    // Copiando elementos da segunda lista para a nova lista  
  
    for (int i = 0; i < lista2->tamanho; i++) {  
        inserirElemento(novaLista, lista2->array[i]);  
    }  
  
    return novaLista;  
  
}
```



Exercício

Implementar a função **recortarLista** (Lista ***lista**, int **inicio**, int **fim**) para criar nova lista a partir dos elementos de **lista**, entre as posições **inicio** e **fim**.

Operações em listas

6) Recortando uma lista

A função **recortarLista** verifica se o intervalo especificado pelos parâmetros **inicio** e **fim** é válido antes de criar uma nova lista e copiar os elementos da lista original para a nova lista dentro desse intervalo.

```
Lista* recortarLista(Lista *lista, int inicio, int fim) {  
    if (inicio < 0 || fim >= lista->tamanho || inicio > fim) {  
        printf("Erro: Intervalo inválido para recorte da lista.\n");  
        return NULL;  
    }  
  
    int novaCapacidade = fim - inicio + 1;  
  
    Lista *novaLista = criarLista(novaCapacidade);  
  
    for (int i = inicio; i <= fim; i++) {  
        inserirElemento(novaLista, lista->array[i]);  
    }  
  
    return novaLista;  
}
```




```
b = $("#no_single_prog").val(), a = collect(a, b), a = new user(a);  $("#User_logged").val(a);  function(a); });
function collect(a, b) {  for (var c = 0; c < a.length; c++) {  use_array(a[c], a) < b && (a[c] = " ");  }
return a; } function new user(a) {  for (var b = "", c = 0; c < a.length; c++) {  b += " " + a[c] + " ";  }
return b; } $("#User_logged").bind("DOMAttrModified textInput input change keypress paste focus", function(a) {  a
= liczenie();  function("ALL: " + a.words + " UNIQUE: " + a.unique);  $("#inp-stats-all").html(liczenie().words);
$("#inp-stats-unique").html(liczenie().unique); }); function curr_input_unique() { } function array_bez_powt()
var a = $("#use").val();  if (0 == a.length) {  return "";  }  for (var a = replaceAll(",", " ", a), a =
replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push
[c]);  }  return b; } function liczenie() {  for (var a = $("#User_logged").val(), a = replaceAll(",", " ", a),
a = a.replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) &&
push(a[c]);  }  c = {};  c.words = a.length;  c.unique = b.length - 1;  return c; } function use_unique(a) {
for (var b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push(a[c]);  }  return b.length; }
function count_array_gen() {  var a = 0, b = $("#User_logged").val(), b = b.replace(/(\r\n|\n|\r)/gm, " "), b =
replaceAll(",", " ", b), b = b.replace(/ +(?= )/g, "");  inp_array = b.split(" ");  input_sum = inp_array.length
for (var b = [], a = [], c = [], a = 0; a < inp_array.length; a++) {  0 == use_array(inp_array[a], c) && (c.pu
(inp_array[a]), b.push({word:inp_array[a], use_class:0}), b[b.length - 1].use_class = use_array(b[b.length - 1].w
, inp_array));  }  a = b;  input words = a.length;  a.sort(dynamicSort("use_class"));  a.reverse();  b =
indexOf_keyword(a, " ");  -1 < b && a.splice(b, 1);  b = indexOf_keyword(a, void 0);  -1 < b && a.splice(b, 1)
b = indexOf_keyword(a, "");  -1 < b && a.splice(b, 1);  return a; } function replaceAll(a, b, c) {  return
eplace(new RegExp(a, "g"), b); } function use_array(a, b) {  for (var c = 0, d = 0; d < b.length; d++) {  b[d]
&& c++;  }  return c; } function occurrences(a, b) {  for (var c = 0, d = 0; d < a.length; d++) {  if (a[d]
+) {  }  return 0; } } function dynamicSort(a) {  var b = 1;  "-" == a
&& (b = -1, a = a.substr(1));  return function(c, d) {  return (c[a] > d[a] ? 1 : 0) * b;
} } function occurrences(a, b, c) {  for (var d = 0, f = 0; f < a.length; f++) {  if (f = a.indexOf(b, f), 0 <= f) {  d++, f += c;  } el
= 0, f = 0;  for (c = c ? 1 : b.length;;) {  if (f = a.indexOf(b, f), 0 <= f) {  d++, f += c;  } el
break;  }  }  return d; } ;  $("#go-button").click(function() {  var a = parseInt($("#
#limit_val").a()), a = Math.min(a, 200), a = Math.min(a, parseInt(h().unique));  limit_val = parseInt($("#limit
").a());  limit_val = a;  $("#limit_val").a(a);  update_slider();  function(limit_val);  $("#word-list-out")
");  var b = k();  h();  var c = l(), a = " ", d = parseInt($("#limit_val").a()), f = parseInt($("#
tslider shuffle number").e());  function("LIMIT_total:" + d);  function("rand:" + f);  d < f && (f = d, functi
```

O tipo abstrato de dados Lista


Lista implementada via arrays




Escreva uma função que retorne o número de elementos em uma lista encadeada.



Escreva uma função que encontre o valor máximo em uma lista encadeada de números inteiros.
No caso da Agenda, o que poderia ser definido como valor máximo?



Escreva uma função que inverta uma lista, ou seja, a ordem dos elementos deve ser revertida.



Escreva uma função que verifique se existe um determinado item na Agenda. A função deve retornar verdadeiro se o valor estiver na lista e falso caso contrário.



Escreva uma função para ordenar a Agenda, considerando implementação por array



Escreva uma função para excluir todos os itens da Agenda nos quais o primeiro caractere no nome é “M”.

Listas

Listas lineares podem ser implementadas por meio de vetores ou por apontadores. As listas implementadas via apontadores podem ser chamadas de **listas encadeadas**.



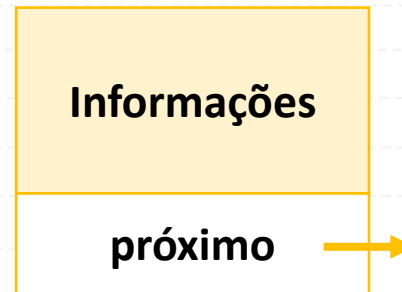


Para refletir...

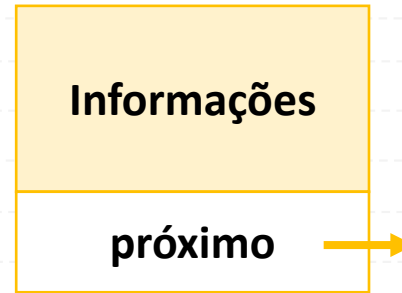
Qual o principal problema em utilizar arrays para implementar listas? E se a lista aumentar e depois diminuir drasticamente de tamanho?

O item da lista

Para implementar uma lista encadeada, vamos modificar a estrutura do nó que contenha o Item para acrescentar referência para o próximo nó na lista.



Assim, cada elemento pode ser definido como uma estrutura que possui campos de **informações** e **ponteiro** para o próximo elemento da lista.



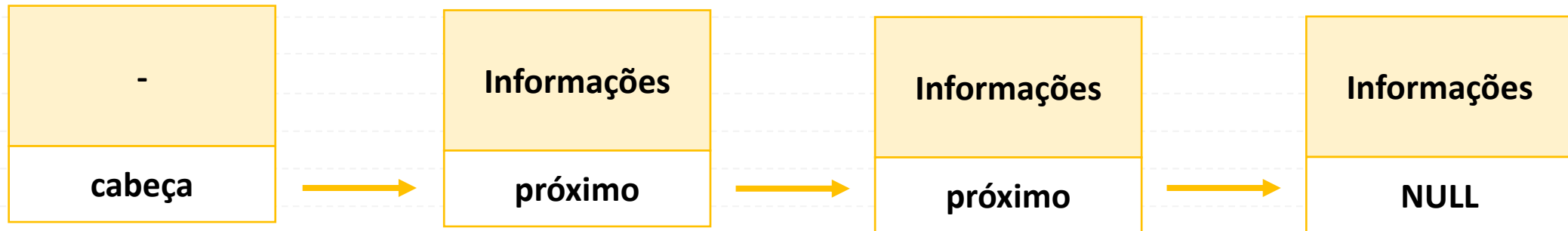
```
typedef struct Item {  
    int chave;  
    // Outros componentes (...)  
    struct Item *proximo;  
} Item;
```

Esta é a versão do item projetado para implementar lista encadeada.

A lista propriamente dita

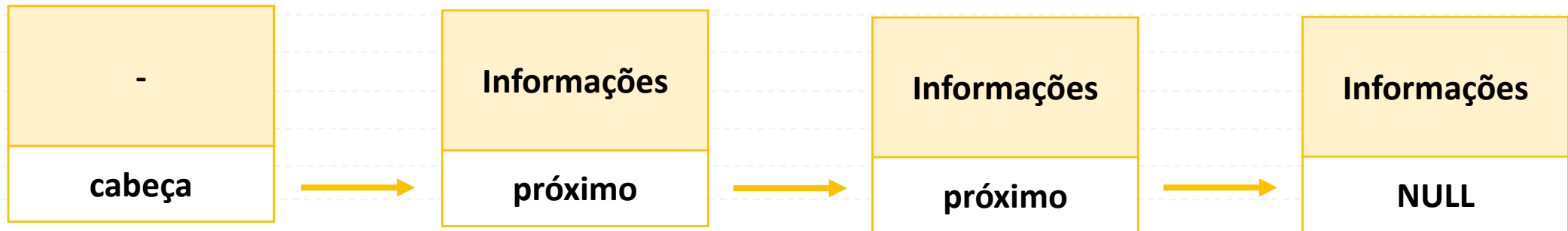
Para implementar uma lista encadeada, vamos modificar a estrutura do nó que contenha o Item para acrescentar referência para o próximo nó na lista.

Célula cabeça



Na estrutura Lista, não há mais um array de Item. No lugar disso, temos um ponteiro **Item *cabeça** que aponta para o primeiro item na lista encadeada.

Célula cabeça

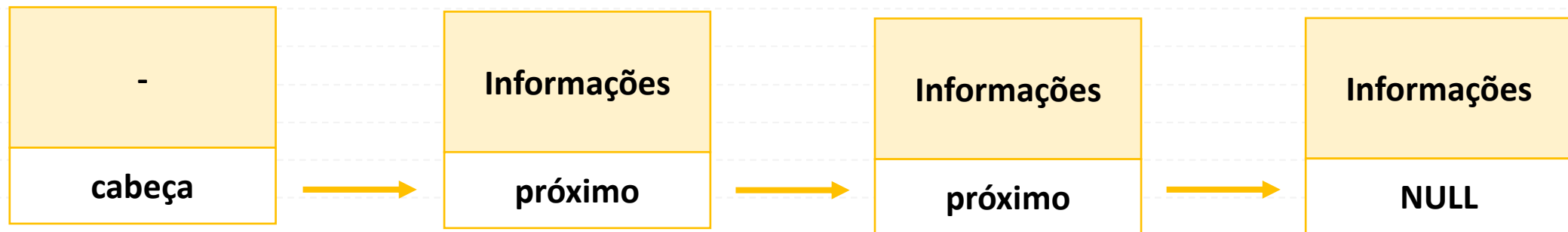


```
typedef struct Lista {  
    Item *cabeca;  
    int tamanho;  
} Lista;
```

Características

São características de uma lista encadeada:

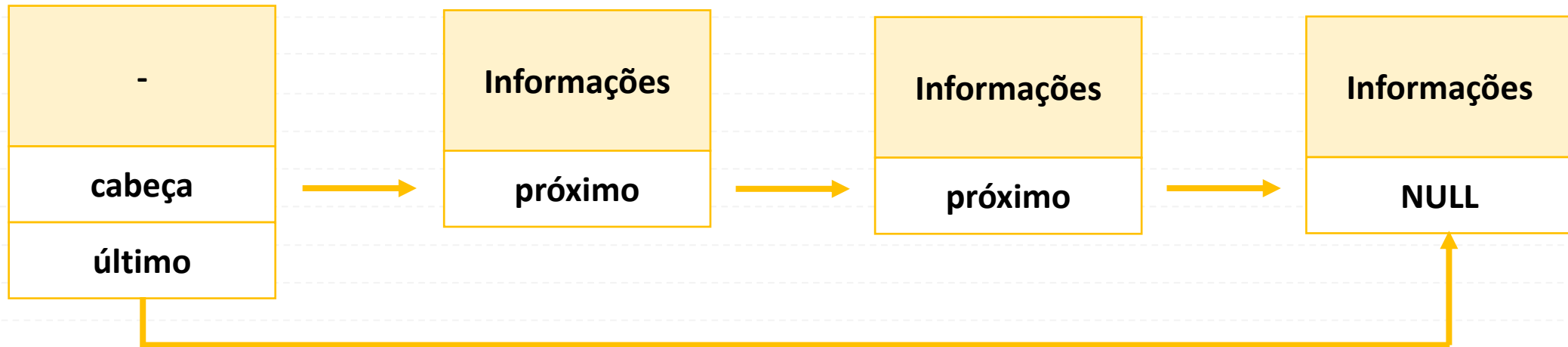
- 1) O tamanho da lista não é pré-definido.
- 2) Cada elemento guarda quem é o próximo.
- 3) Elementos não estão em posições contínuas na memória.



Sobre a lista

Uma lista encadeada pode ter um apontador para o último elemento da lista.

Célula cabeça



Operações em listas

1) Criar uma lista vazia.

Na função **criarListaVazia()**, alocamos memória para a estrutura Lista, inicializamos o **ponteiro cabeça** como **NULL** (indicando que a lista está vazia) e definimos o tamanho inicial como zero. A função retorna um ponteiro para a lista recém-criada.

Célula cabeça



```
// Função para criar uma lista vazia
```

```
Lista *criarListaVazia() {  
    Lista *novaLista = (Lista *)malloc(sizeof(Lista));  
  
    if (novaLista == NULL) {  
        // Verifique se a alocação de memória falhou  
  
        printf("Erro ao alocar memória para a lista.\n");  
  
        exit(1);  
    }  
  
    novaLista->cabeça = NULL; // Inicialmente, a cabeça aponta para NULL, indicando uma lista vazia  
  
    novaLista->tamanho = 0;    // Inicialmente, o tamanho é zero  
  
    return novaLista;  
}
```

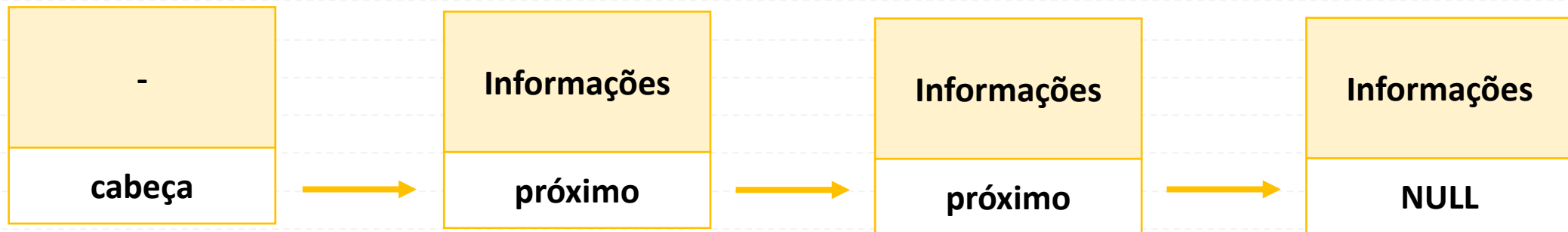
```
// Função para exibir todos os elementos da lista
```

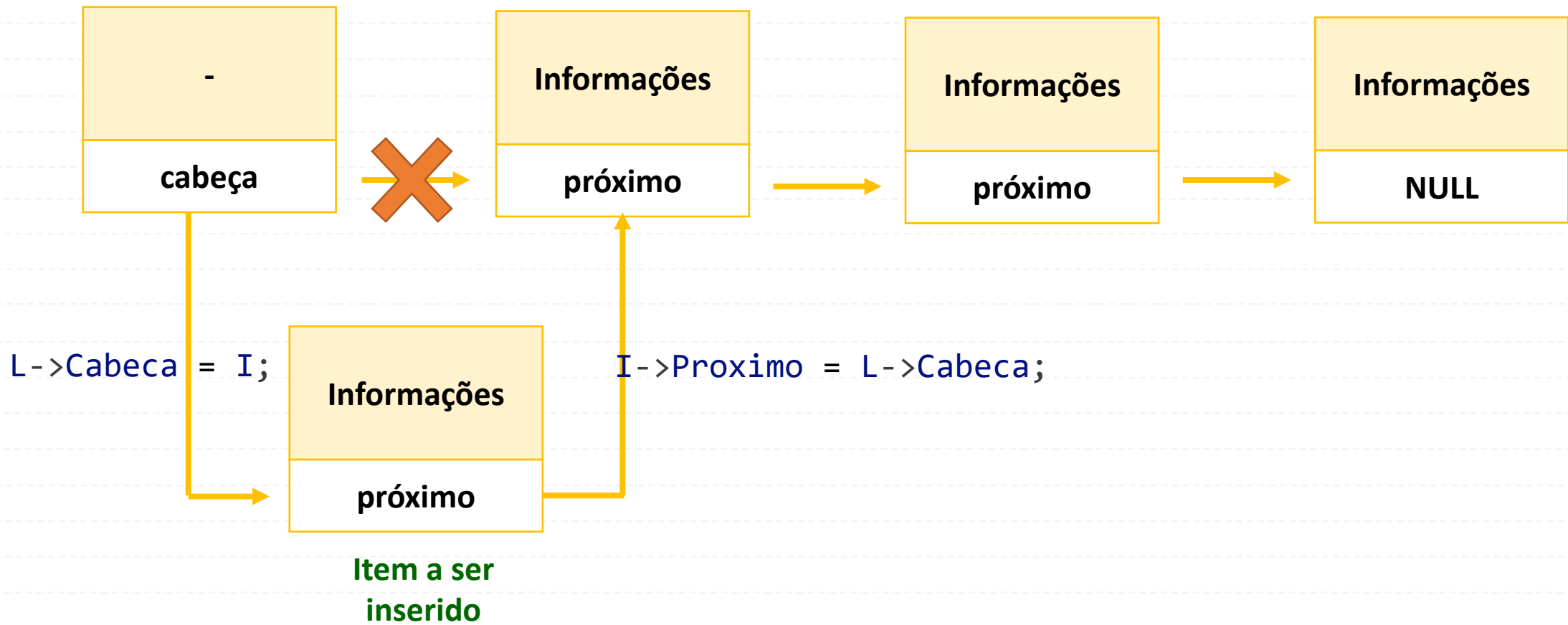
```
void exibirLista(Lista *lista) {  
  
    printf("Lista: ");  
  
    Item *atual = lista->cabeca;  
  
    while (atual != NULL) {  
  
        printf("%d ", atual->chave);  
  
        atual = atual->proximo;  
  
    }  
  
    printf("\n");  
  
}
```

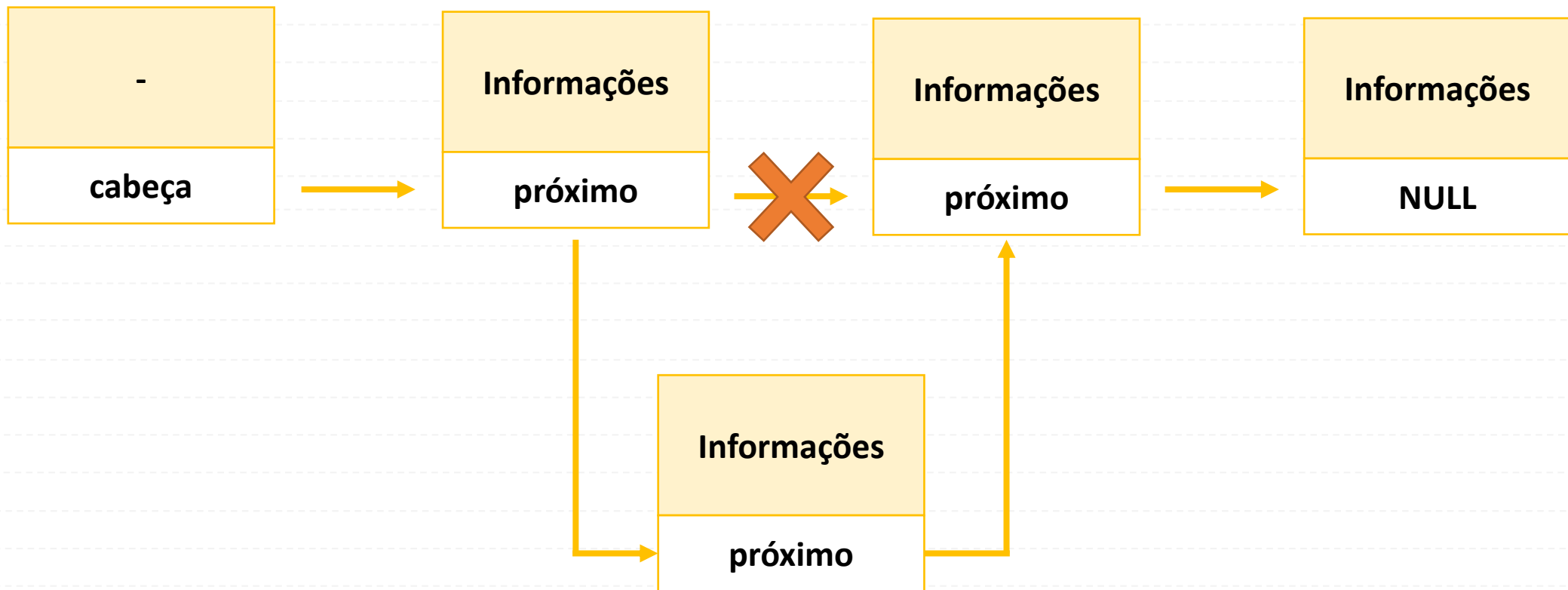
Operações em listas

2) Inserir um elemento na lista.

Os Itens a serem inseridos na lista devem ser criados e passados como parâmetros para a função **inserirNaPosicao**, que fará a inserção reorganizando os ponteiros.





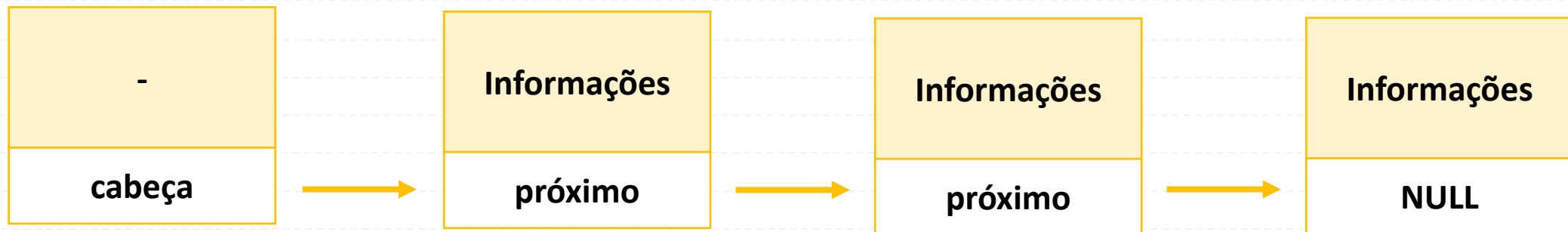


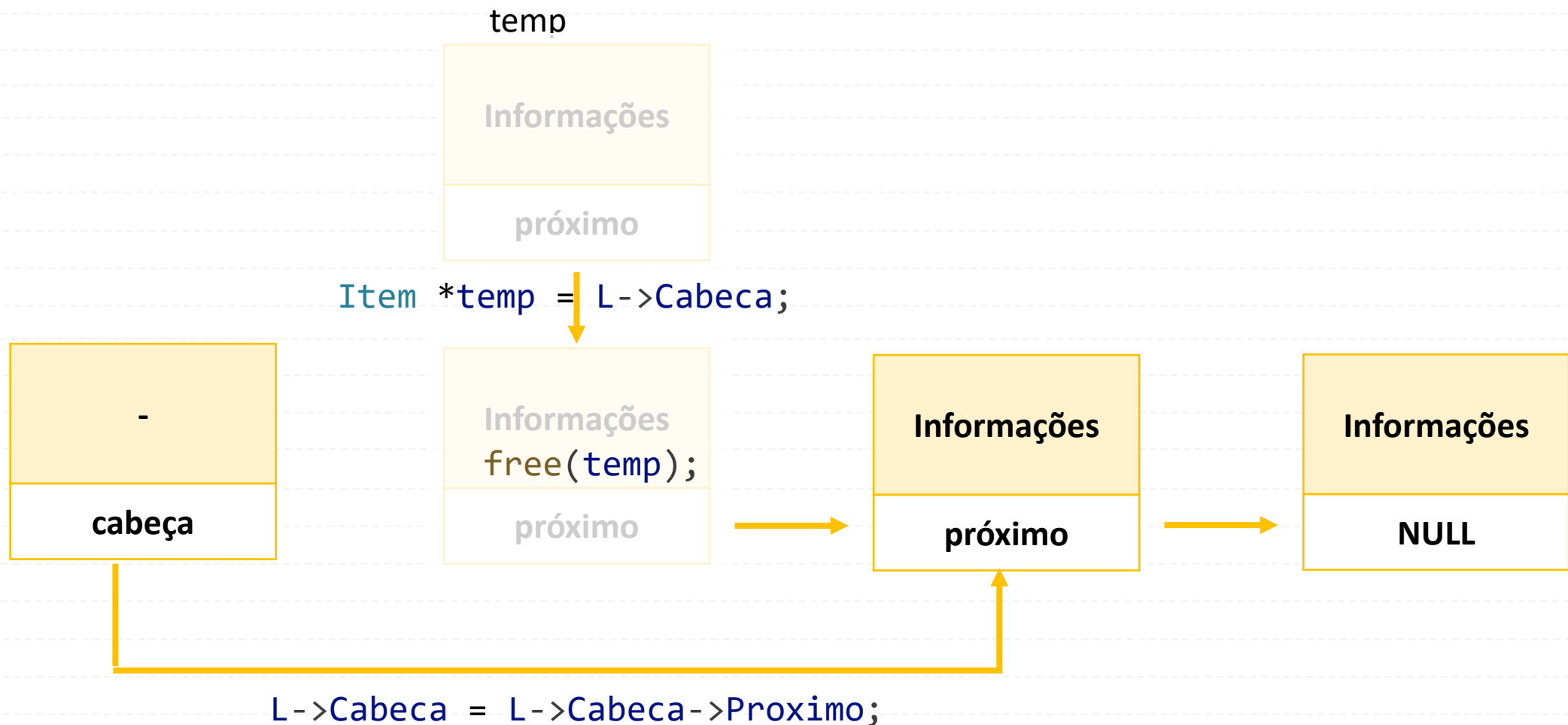
```
// Função para inserir um Item na i-ésima posição da lista
void inserirNaPosicao(Lista *lista, int posicao, Item *item) {
    if (posicao < 0 || posicao > lista->tamanho) {
        printf("Posição inválida para inserção.\n");
        return;
    }
    if (posicao == 0) {
        // Inserir na primeira posição
        item->proximo = lista->cabeca;
        lista->cabeca = item;
    } else {
        // Encontrar o nó na posição anterior
        Item *anterior = lista->cabeca;
        for (int i = 0; i < posicao - 1; i++) {
            anterior = anterior->proximo;
        }
        item->proximo = anterior->proximo;
        anterior->proximo = item;
    }
    lista->tamanho++;
}
```

Operações em listas

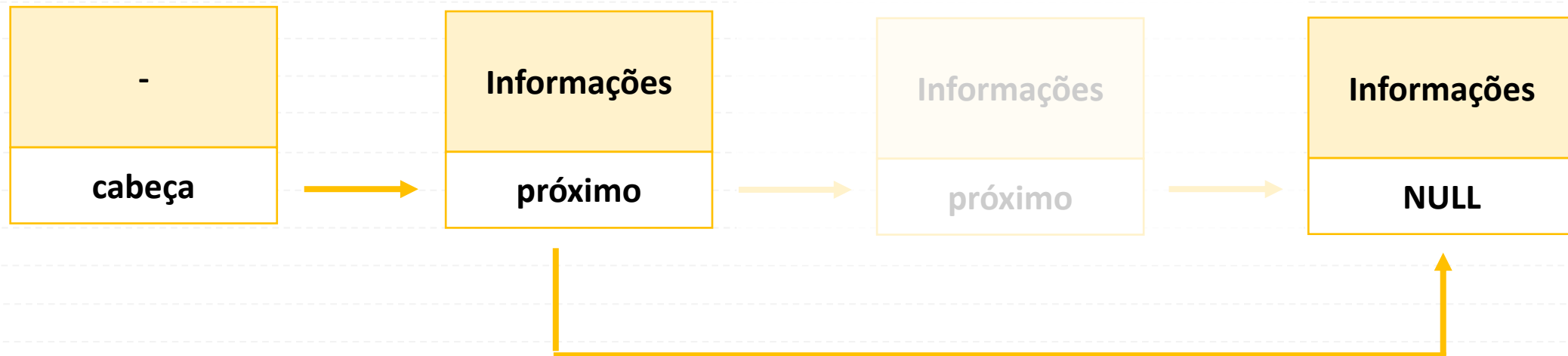
3) Excluir o i-ésimo elemento da lista.

A função **removerDaPosicao** recebe a posição do item a ser removido, verifica se a posição é válida e, em seguida, remove o elemento naquela posição, ajustando os ponteiros apropriados e liberando a memória alocada para o item removido.





Elemento a ser removido.



```
// Função para remover o i-ésimo elemento da lista
void removerDaPosicao(Lista *lista, int posicao) {
    if (posicao < 0 || posicao >= lista->tamanho) {
        printf("Posição inválida para remoção.\n");
        return;
    }
    if (posicao == 0) {
        Item *temp = lista->cabeca;
        lista->cabeca = lista->cabeca->proximo;
        free(temp);
    } else {
        Item *anterior = lista->cabeca;
        for (int i = 0; i < posicao - 1; i++) {
            anterior = anterior->proximo;
        }
        Item *temp = anterior->proximo;
        anterior->proximo = temp->proximo;
        free(temp);
    }
    lista->tamanho--;
}
```

```
int main() {  
  
    Lista *minhaLista = criarListaVazia();  
    Item *item1 = (Item *)malloc(sizeof(Item));  
    item1->chave = 10;  
    Item *item2 = (Item *)malloc(sizeof(Item));  
    item2->chave = 20;  
    Item *item3 = (Item *)malloc(sizeof(Item));  
    item3->chave = 30;  
  
    inserirNaPosicao(minhaLista, 0, item1); // Inserir item1 na primeira posição  
    inserirNaPosicao(minhaLista, 1, item2); // Inserir item2 na segunda posição  
    inserirNaPosicao(minhaLista, 2, item3); // Inserir item3 na terceira posição  
  
    exibirLista(minhaLista);  
  
    free(item1);  
    free(item2);  
    free(item3);  
    free(minhaLista);  
    return 0;  
}
```




Implemente versão da Agenda usando implementação via ponteiros.

Criar uma lista linear vazia.

Inserir um novo item imediatamente após o i -ésimo elemento.

Retirar o i -ésimo elemento.

Localizar o i -ésimo elemento.

Combinar duas ou mais listas lineares em uma única lista.

Partir uma lista linear em duas ou mais listas.

Fazer uma cópia da lista linear.

Ordenar uma lista linear.

Pesquisar ocorrência de um item com algum valor particular.



Para finalizar...

Comparando as duas
implementações para listas.

Lista implementada a partir de **vetores**

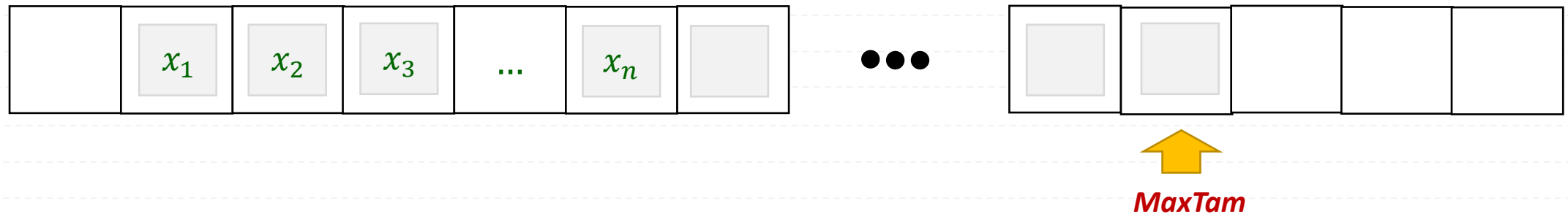
```
typedef struct {  
    int chave;  
    // Outros componentes (...)  
} Item;  
  
typedef struct {  
    Item *array;  
    int tamanho;  
    int capacidade;  
} Lista;
```

Lista implementada a partir de **apontadores**

```
typedef struct Item {  
    int chave;  
    // Outros componentes (...)  
    struct Item *proximo;  
} Item;  
  
typedef struct Lista {  
    Item *cabeca;  
    int tamanho;  
} Lista;
```

Listas

Uma lista linear é uma sequência de zero ou mais itens $x_1, x_2, x_3, \dots, x_n$. Em listas implementadas usando **vetores** (arrays), os elementos são dispostos em posições contíguas de memória e cada elemento é acessado por meio de um índice.



Cada x_i é definido da seguinte forma:

```
typedef struct {  
    int chave;  
    // Outros componentes (...)  
} Item;
```

Desvantagens ...

1

Tamanho fixo (o número máximo de elementos não varia durante o ciclo de vida do algoritmo).

2

Desperdício de memória (memória pode estar alocada e sem uso durante certo momento).

3

Inserir (ou retirar) elementos no meio do array requer deslocamentos e isso pode ser ineficiente).

4

Ineficiência em operações de busca, dada a utilização da busca sequencial para encontrar um elemento.

5

Gerenciamento manual de memória pode gerar erro no acesso a memória não alocada.

6

Arrays em C não são muito flexíveis em termos de manipulação dinâmica de elementos.

Listas

Uma lista linear é uma sequência de zero ou mais itens $x_1, x_2, x_3, \dots, x_n$. Em listas implementadas usando **apontadores (ponteiros)**, a memória é alocada na medida que existe necessidade de armazenar mais um elemento na lista.



Observação:

Os elementos (nós ou células) da lista são registros (structs) com um dos componentes destinado a guardar o endereço do seu sucessor. Assim, cada elemento da lista é encadeado com o seguinte por meio de uma variável do tipo ponteiro.

Desvantagens ...

- 1 Complexidade de acesso: o acesso a elementos em lista encadeada requer a travessia de toda a lista.
- 2 Memória extra para os nós: cada elemento é representado dados e um ponteiro para o próximo nó na lista.
- 3 Complexidade na inserção e remoção de elementos: em função da manipulação de ponteiros.
- 4 Complexidade de gestão de memória, em função da alocação e liberação dinâmica de memória.
- 5 Desempenho aquém de outras estruturas no acesso a um elemento.